CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 4

In this assignment you will create a calculator for performing matrix operations that exploits the (expected) sparseness of its matrix operands. An $n \times n$ square matrix is said to be *sparse* if the number of non-zero entries (abbreviated NNZ) is small compared to the total number of entries n^2. The result will be a C program capable of performing fast matrix operations, even on very large matrices, provided that they are sparse.

Given $n \times n$ matrices *A* and *B*, their product $C = A \cdot B$ is the $n \times n$ matrix whose ij^{th} entry is given by

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Thus the element in the i^{th} row and j^{th} column of *C* is the vector dot product of the i^{th} row of *A* with the j^{th} column of *B*. If we consider addition and multiplication of real numbers to be our basic operations, then the above formula can be computed in time $\Theta(n^3)$, which is impractical for matrix sizes *n* of more than a few thousand. If it so happens that *A* and *B* are sparse, then a great many of these arithmetic operations involve adding or multiplying by zero, hence are unnecessary.

The sum *S*, and difference *D*, of *A* and *B* are the $n \times n$ matrices having ij^{th} entries:

$$S_{ij} = A_{ij} + B_{ij} \qquad \text{and} \qquad D_{ij} = A_{ij} - B_{ij}$$

The scalar product of a real number *x* with *A* is denoted xA, and has ij^{th} entry $(xA)_{ij} = x \cdot A_{ij}$. The transpose of *A*, denoted A^T, is the matrix whose ij^{th} entry is the ji^{th} entry of *A*: $(A^T)_{ij} = A_{ji}$. Thus, the rows of *A* are the columns of A^T, and the columns of *A* are the rows of A^T. Each of these operations can be computed in time $\Theta(n^2)$, and just as for multiplication, their cost can be improved upon significantly when *A* and *B* are sparse.

As one would expect, the cost of a matrix operation depends heavily on the choice of data structure used to represent its matrix operands. There are several ways to represent a square matrix with real entries. The standard approach is to use a 2-dimensional $n \times n$ array of doubles. The advantage of this representation is that all of the above matrix operations have a straight-forward implementation using nested loops. This project will use a very different representation however. Here you will represent a matrix as a 1-dimensional array of Lists. Each List will represent one row of the Matrix, but only the non-zero entries will be stored. Therefore List elements must store, not just the matrix entries, but the column index in which those entries reside. For example, the matrix below would have the following representation as an array of Lists.

$$M = \begin{bmatrix} 1.0 & 0.0 & 2.0 \\ 3.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 5.0 \end{bmatrix} \qquad \text{Array of Lists: } \left[\begin{array}{lll} 1: & (1, 1.0) & (3, 2.0) \\ 2: & (1, 3.0) & \\ 3: & (2, 4.0) & (3, 5.0) \end{array}\right.$$

This method obviously results in a substantial space savings when the Matrix is sparse. In addition, the standard matrix operations defined above can be performed more efficiently on sparse matrices. As you will see though, the matrix operations are much more difficult to implement using this representation. The trade-off then, is a gain in space and time efficiency for sparse matrices, at the expense of more complicated algorithms for performing standard matrix operations. Designing these algorithms in terms of our List ADT operations will constitute the majority of the work you do on this assignment.

It will be necessary to first make some minor changes to your List ADT from pa1. First you must convert it from a List of ints to a List of generic pointers. This entails changing certain field types, declaration statements, method parameters, and return types from `int` to `void*`. The objects referred to by these List elements will be defined in the Matrix ADT specified below. Second, it will be necessary to eliminate the List operations equals() and copyList(). The problem with these functions is that the List no longer knows what it is a list of, and therefore can only perform "shallow" versions of these operations, i.e. compare or copy pointers, not what they point to. For the same reason, function printList() is no longer required (but you may wish to include it for diagnostic purposes only). All other List operations from pa1 will be retained. The (optional) function cat() may also be included.

A "typical loop" in a client of List (as previously illustrated on page 3 of the pa1 project description) might now appear as

```
type x;  // Here "type" is whatever data type the void
         // pointers in List are pointing to, known by
         // the client but not by List.
moveFront(L);
while( index(L)>=0 ){
   // Get current element, cast as the right kind of
   // pointer, follow the pointer, assign its value to x.
   x = *(type*)get(L);

   // do something with x, then
   moveNext(L);
}
```

Be sure to test this modified List ADT before you try to implement the Matrix ADT defined below. A (very weak) test of the modified List ADT called ListClient.c will be posted under the examples section of the class webpage.

File Formats

The top level client module for this project will be called Sparse.c. It will take two command line arguments giving the names of the input and output files, respectively. The input file will begin with a single line containing three integers *n*, *a* and *b*, separated by spaces. The second line will be blank, and the following *a* lines will specify the non-zero entries of an $n \times n$ matrix *A*. Each of these lines will contain a space separated list of three numbers: two integers and a double, giving the row, column, and value of the corresponding matrix entry. After another blank line, there will follow *b* lines specifying the non-zero entries of an $n \times n$ matrix *B*. For example, the two matrices

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

are encoded by the following input file:

```
3 9 5

1 1 1.0
1 2 2.0
1 3 3.0
2 1 4.0
2 2 5.0
2 3 6.0
3 1 7.0
3 2 8.0
3 3 9.0

1 1 1.0
1 3 1.0
3 1 1.0
3 2 1.0
3 3 1.0
```

Your program will read an input file as above, initialize and build the Array-of-Lists representation of the matrices A and B, then calculate and print the following matrices to the output file: A, B, $(1.5)A$, $A+B$, $A+A$, $B-A$, $A-A$, A^T, AB and B^2. The output file format is illustrated by the following example, which corresponds to the above input file.

```
A has 9 non-zero entries:
1: (1, 1.0) (2, 2.0) (3, 3.0)
2: (1, 4.0) (2, 5.0) (3, 6.0)
3: (1, 7.0) (2, 8.0) (3, 9.0)

B has 5 non-zero entries:
1: (1, 1.0) (3, 1.0)
3: (1, 1.0) (2, 1.0) (3, 1.0)

(1.5)*A =
1: (1, 1.5) (2, 3.0) (3, 4.5)
2: (1, 6.0) (2, 7.5) (3, 9.0)
3: (1, 10.5) (2, 12.0) (3, 13.5)

A+B =
1: (1, 2.0) (2, 2.0) (3, 4.0)
2: (1, 4.0) (2, 5.0) (3, 6.0)
3: (1, 8.0) (2, 9.0) (3, 10.0)

A+A =
1: (1, 2.0) (2, 4.0) (3, 6.0)
2: (1, 8.0) (2, 10.0) (3, 12.0)
3: (1, 14.0) (2, 16.0) (3, 18.0)

B-A =
1: (2, -2.0) (3, -2.0)
2: (1, -4.0) (2, -5.0) (3, -6.0)
3: (1, -6.0) (2, -7.0) (3, -8.0)

A-A =
```

```
Transpose(A) =
1: (1, 1.0) (2, 4.0) (3, 7.0)
2: (1, 2.0) (2, 5.0) (3, 8.0)
3: (1, 3.0) (2, 6.0) (3, 9.0)

A*B =
1: (1, 4.0) (2, 3.0) (3, 4.0)
2: (1, 10.0) (2, 6.0) (3, 10.0)
3: (1, 16.0) (2, 9.0) (3, 16.0)

B*B =
1: (1, 2.0) (2, 1.0) (3, 2.0)
3: (1, 2.0) (2, 1.0) (3, 2.0)
```

Notice that the rows are to be printed in column sorted order, and zero rows are skipped altogether. A zero matrix will cause no output to be printed, as seen by the matrix $A - A$ above. Note that, unlike the output file, the input file may give matrix entries in any order.

Matrix ADT Specifications

In addition to the main program Sparse.c and the altered List.c from pa1, you will implement a Matrix ADT in a file called Matrix.c. This ADT will contain a private inner struct (similar to Node in your List ADT) encapsulating the column and value information corresponding to a matrix entry. You can give this inner struct any name you wish, but I will refer to it here as Entry. Thus Entry is a pointer to a struct called EntryObj that has two fields of types int and double respectively. Your Matrix ADT will represent a matrix as an array of Lists of Entries. Since Entry is itself a pointer, it can be assigned to a field of type `void*`. It is required that each List of Entries be maintained in column sorted order. Your Matrix ADT will export the following operations.

```
// newMatrix()
// Returns a reference to a new nXn Matrix object in the zero state.
Matrix newMatrix(int n)

// freeMatrix()
// Frees heap memory associated with *pM, sets *pM to NULL.
void freeMatrix(Matrix* pM);


// Access functions

// size()
// Return the size of square Matrix M.
int size(Matrix M);

// NNZ()
// Return the number of non-zero elements in M.
int NNZ(Matrix M);

// equals()
// Return true (1) if matrices A and B are equal, false (0) otherwise.
int equals(Matrix A, Matrix B);
```

```
// Manipulation procedures

// makeZero()
// Re-sets M to the zero Matrix state.
void makeZero(Matrix M);

// changeEntry()
// Changes the ith row, jth column of M to the value x.
// Pre: 1<=i<=size(M), 1<=j<=size(M)
void changeEntry(Matrix M, int i, int j, double x);


// Matrix Arithmetic operations

// copy()
// Returns a reference to a new Matrix object having the same entries as A.
Matrix copy(Matrix A);

// transpose()
// Returns a reference to a new Matrix object representing the transpose
// of A.
Matrix transpose(Matrix A);

// scalarMult()
// Returns a reference to a new Matrix object representing xA.
Matrix scalarMult(double x, Matrix A);

// sum()
// Returns a reference to a new Matrix object representing A+B.
// pre: size(A)==size(B)
Matrix sum(Matrix A, Matrix B);

// diff()
// Returns a reference to a new Matrix object representing A-B.
// pre: size(A)==size(B)
Matrix diff(Matrix A, Matrix B);

// product()
// Returns a reference to a new Matrix object representing AB
// pre: size(A)==size(B)
Matrix product(Matrix A, Matrix B);

// printMatrix()
// Prints a string representation of Matrix M to filestream out. Zero rows
// are not printed.  Each non-zero row is represented as one line consisting
// of the row number, followed by a colon, a space, then a space separated
// list of pairs "(col, val)" giving the column numbers and non-zero values
// in that row. The double val will be rounded to 1 decimal point.
void printMatrix(FILE* out, Matrix M);
```

It is required that your program perform these operations efficiently. Let n be the number of rows in A, and let a and b denote the number of non-zero entries in A and B respectively. Then the worst case run times of the above functions should have the following asymptotic growth rates.

```
changeEntry(A, i, j, x):    Θ(a)
copy(A):                    Θ(n + a)
transpose(A):               Θ(n + a)
scalarMult(x, A):           Θ(n + a)
sum(A, B):                  Θ(n + a + b)
diff(A, B):                 Θ(n + a + b)
product(A, B):              Θ(n^2 + a · b)
```

See the handout entitled *Algorithm Runtime and Efficiency* posted on the class webpage for an explanation of the Θ notation above. It will be helpful to include a private function in Matrix.c with signature

```
double vectorDot(List P, List Q)
```

that computes the vector dot product of two matrix rows represented by Lists P and Q. Use this function together with function `transpose()` to help implement `product()`. Similar helper functions for the operations `sum()` and `diff()` will also be very useful, and will be discussed in class.

What to Turn In

Your project will be structured in three files: Sparse.c, Matrix.c, and List.c (together with header files Matrix.h and List.h). The main program Sparse.c, will handle the input and output files and is the client of the Matrix ADT, which is itself a client of the modified List ADT. Note that Sparse is not itself a direct client of List, since it need not call any List operations. You will also write separate client modules `ListTest.c` and `MatrixTest.c` to test the List and Matrix ADTs in isolation. Students often ask what should be the contents of these test files. In each case, include enough calls to ADT operations to convince the grader that you did in fact test your List and Matrix ADT modules in isolation before using them in the larger project. The best way to do this is to *actually use them for this purpose*. At minimum they should call every public function in their respective ADT modules at least once. The webpage link Examples/pa4 will contain files MatrixClient.c and ListClient.c. These should be considered to be weak tests of your Matrix and List modules respectively, and are not adquate for your testing purposes. A number of matched pairs of input/output files will also be included, along with a Python script for creating random input files.

Also submit a `README.md` file and a `Makefile` that creates an executable file called `Sparse`. (A Makefile is also included in Examples/pa4 which you may alter as you see fit.) Thus in all, nine files will be submitted.

`Sparse.c`	written by you
`Matrix.c`	" " "
`Matrix.h`	" " "
`MatrixTest.c`	" " "
`List.c`	" " "
`List.h`	" " "
`ListTest.c`	" " "
`README.md`	" " "
`Makefile`	provided, change if you like

Push these files to the pa4 folder in your git repository on git.ucsc.edu by the due date. As always, start early and ask questions.